

The Monad.Reader Issue 18

by Julian Porter <julian.porter@porternet.org>
and Tomas Petricek <tomas.petricek@cl.cam.ac.uk>
and Douglas M. Auclair <dauclair@hotmail.com>

July 5, 2011



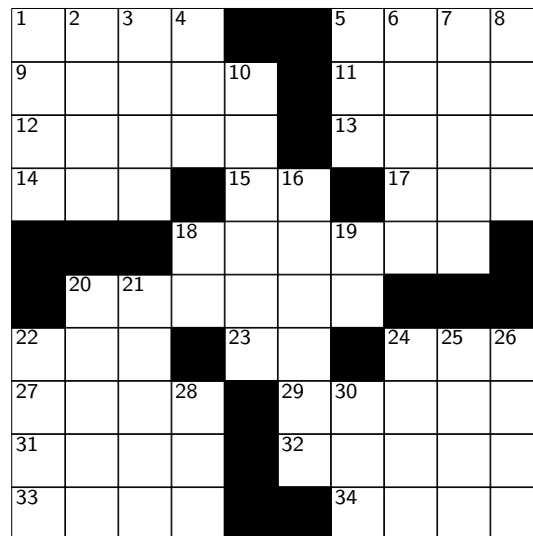
Brent Yorgey, editor.

Contents

Brent Yorgey	
Editorial	3
Julian Porter	
MapReduce as a Monad	5
Tomas Petricek	
Fun with Parallel Monad Comprehensions	17
Douglas M. Auclair	
Attributed Variables: Their Uses and One Implementation	43

Editorial

by Brent Yorgey (byorgey@cis.upenn.edu)



Across

- 1 Quaff
- 5 Improvised vessel
- 9 Regions
- 11 Then again... (abbr.)
- 12 101
- 13 Norway's capital
- 14 Chi follower
- 15 Unix utility
- 17 Kmett or Yang (abbr.)
- 18 "_____ programming solves a vast majority of the World's problems"
- 20 Map partner
- 22 US encryption stan-

dard

- 23 Haskell competitor
- 24 Pi competitor
- 27 GADT part
- 29 Gives off
- 31 False god
- 32 Work out
- 33 Code after colons
- 34 Disorder

Down

- 1 Parallel monad method
- 2 Ages
- 3 Against (prefix)
- 4 German article
- 5 Tigger's pal

- 6 On a voyage
- 7 List catamorphism
- 8 Tank eng. of note
- 10 Reactive metal
- 16 Male relatives
- 18 Unix utility
- 19 ENIAC subj.
- 20 To hand
- 21 Prohibit
- 22 Mine entrance
- 24 Emulate Escher
- 25 Off-roaders
- 26 Applications
- 28 Quaff
- 30 Tattoo subject

MapReduce as a Monad

by Julian Porter <julian.porter@porternet.org>

MapReduce is an increasingly popular paradigm for building massively parallel applications. It originated in ideas from functional programming, but soon migrated to a more imperative approach. In the process, popular implementations have become massively complicated to run. For example, HADOOP requires considerable detailed knowledge on the part of the application's author to ensure that their code plugs properly into the MapReduce framework.

I show how MapReduce can be implemented as a form of monad in Haskell. Why a monad? First, it means processing code is purely about processing; it need not know anything about the MapReduce framework. Second, MapReduce reduces to repeated composition of monadic functions with \gg , so the programmer can trivially mix MapReduce with other paradigms. Third, it allows a massive generalisation of MapReduce. And fourth, it's an interesting exercise in monadic programming.

This paper describes the basic approach to MapReduce and discusses performance of a multi-threaded application running on a single machine. In a forthcoming article I will demonstrate how this approach can be generalised still further and made to work on a distributed cluster.

About MapReduce

MapReduce consists of two stages, named **map** and **reduce**. In the map stage, the following happens:

1. A central unit takes data in the form of value/key pairs and divides it into a number of chunks, which it distributes to one or more **mappers**.
2. Each mapper transforms its input list of value/key pairs into an output list of value/key pairs.

In the reduce stage, the following happens:

1. A central unit sorts the output of the map stage by key, dividing it into chunks, one for each unique key value in the set, and distributes the chunks to **reducers**.
2. Each reducer transforms its input value/key pairs into an output list of value/key pairs.

This process is repeated until the desired result is achieved.

The mapper and reducer can be many to many functions, so one input pair may correspond to no output pairs or to many. Also, the type of the output from these functions need not be the same as that of the input.

In practice, there is an intermediate stage called **partitioning**, which is a second reduce stage. Therefore, anything we say about reduce applies to it as well. Note that as my approach removes the distinction between map and reduce, partitioning fits in naturally as just another such step.

The idea

Transformer functions

Consider the reduce stage. It collects data from the map stage and then partitions it into a number of sets, one per key value. Then the reducer is applied to each set separately, producing new value/key pairs for the next step. Restating this, what happens is that for each key value k , reduce extracts the pairs with key value equal to k and then applies the reducer to them:

$$\begin{aligned} \text{reduce}F &:: (Eq\ b) \Rightarrow b \rightarrow ([s] \rightarrow [(s', c)]) \rightarrow ([(s, b)] \rightarrow [(s', c)]) \\ \text{reduce}F\ k\ \text{reducer} &= (\lambda ss \rightarrow \text{reducer}\ \$\ \text{partition}\ k\ ss) \end{aligned}$$

where

$$\text{partition}\ k\ ss = \text{fst}\ \$\ \text{filter}\ (\lambda(-, k') \rightarrow k \equiv k')\ ss$$

The map stage is similar. In the map stage, value/key pairs are distributed randomly across mappers, which then transform their chunk of the data to produce value/key pairs. This is just $\text{reduce}F$ with random key values.

Therefore map and reduce can be fitted within the same conceptual framework, and we can treat the mapper and reducer as instances of a more general concept, the **transformer**, which is a function which takes a list of values and transforms them to produce a list of value/key pairs:

$$[s] \rightarrow [(s', a)]$$

Generalising transformers

There is no reason why $reduceF$ (and its equivalent for map) must take the specific form set out above of a wrapper around a transformer. For a given *transformer* we can define:

$$\begin{aligned} reduceF' &:: (Eq\ a) \Rightarrow a \rightarrow [(s, a)] \rightarrow [(s', b)] \\ reduceF' &= (flip\ reduceF)\ transformer \end{aligned}$$

But the obvious next step is to generalise from $reduceF'$ to any function

$$transformF :: a \rightarrow [(s, a)] \rightarrow [(s', b)]$$

So we have generalised from a function which selects values and feeds them to the transformer to a general function which combines the selection and transformation operations.

Now consider what happens when we compose successive stages. As we have just seen, the outcome of each stage is a map from value/key pairs to value/key pairs. Then in applying the next stage, we take $transformF$ and combine it with that map to obtain a new map from value/key pairs to value/key pairs:

$$\begin{aligned} &([(s, a)] \rightarrow [(s', b)]) \rightarrow (b \rightarrow [(s', b)] \rightarrow [(s'', c)]) \\ &\rightarrow [(s, a)] \rightarrow [(s'', c)] \end{aligned}$$

This is highly suggestive of monadic composition:

$$(Monad\ m) \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$

where m is a function of type $[(s, a)] \rightarrow [(s', b)]$.¹ Suppose we define a type

$$\mathbf{newtype}\ m\ s\ a\ s'\ b = MR\ ([(s, a)] \rightarrow [(s', b)])$$

Then the composition of MapReduce stages becomes:

$$m\ s\ a\ s'\ b \rightarrow (b \rightarrow m\ s'\ b\ s''\ c) \rightarrow m\ s\ a\ s''\ c$$

where s, s', s'' are data types and a, b, c are key types. This certainly looks very much like a “monad” of some description.

¹In fact, this is very similar to the definition of the *State* monad, which is a function of type $(s \rightarrow (s, a))$. The main differences are that the value type changes on application of the function, and also that we need to keep track of the types of the keys.

Using a monad

The goal is then for a single round of generalised MapReduce to be expressible as:

$$state \gg= mapF \gg= reduceF$$

where $mapF$ and $reduceF$ are generalised transformers and $state$ is a monadic value containing the initial state. This approach must contain MapReduce as a special case. Therefore there should be a function

$$wrapMR :: (Eq a) \Rightarrow ([s] \rightarrow [(s', b)]) \rightarrow (a \rightarrow m s a s' b)$$

that wraps a transformer in the monadic framework so that MapReduce can be expressed as:

$$\dots \gg= wrapMR mapper \gg= wrapMR reducer \gg= \dots$$

Aside: do we need all those indices?

Having four indices on m certainly seems rather clumsy. Is there anything we can do to reduce them? For reference, I write out again the composition rule:

$$m s a s' b \rightarrow (b \rightarrow m s' b s'' c) \rightarrow m s a s'' c$$

Note that indices are paired, so objects can be composed if the second value/key pair of the first object equals the first value/key pair of the second.

Therefore, the obvious simplification is to treat value/key pairs as opaque types, so we can rewrite as

$$m p q \rightarrow (q \rightarrow m q r) \rightarrow m p r$$

However, there are two problems which turn out to be related:

1. The first argument of the middle function in MapReduce composition is b , not (s', b) , and so we had to generalise this to q .
2. If we write out this formulation in terms of the functions that comprise m we get:

$$([p] \rightarrow [q]) \rightarrow (q \rightarrow ([q] \rightarrow [r])) \rightarrow ([p] \rightarrow [r])$$

which has lost sight of the distinction between values and keys.

The first problem is survivable, but together with the second it becomes insurmountable. The critical fact about MapReduce is that we take the data set and then, for each key, select a transformer apply it to the whole data set. But now we cannot do that, as all we can see is the opaque value/key pair, and selecting transformers based on key *and* value loses the element of aggregation critical to MapReduce.

There are various (rather ugly) ways we could get round this, e.g.

```
class Keyable kv k | kv → k where
  key :: kv → k
class Monad' m p q where
  return :: q → m p q
  (≫) :: (Keyable q k) ⇒ m p q → (k → m q r) → m p r
```

However, this is clearly just reintroducing the key type by the back door, and is also rather confusing, so we will, regrettably, stick with the four-index version.

Implementation

Monad' and MapReduce

The generalized *Monad'* class is defined as follows:

```
class Monad' m where
  return :: a → m s x s a
  (≫) :: (Eq b) ⇒ m s a s' b → (b → m s' b s'' c) → m s a s'' c
```

The generalised transformers are of a type which is an instance of *Monad'*:

```
newtype MapReduce s a s' b = MR {runMR :: [(s, a)] → [(s', b)]}
```

The monadic operations

The *return* operator is fairly trivial: *return x* just forces the key of the output to be *x*. Composition is the critical part of the design:

```
1  bindMR :: (Eq b) ⇒ MapReduce s a s' b → (b → MapReduce s' b s'' c)
2      → MapReduce s a s'' c
3  bindMR f g = MR $ λs →
4      let
5          fs = runMR f s
```

```

6      gs = map g ∘ nub ∘ map snd $ fs
7      in
8      concat $ map (λg' → runMR g' fs) gs

```

Note that the set of keys is extracted from the output of f in line 5 and used, via g , to generate a set of transformers in line 6, one per key. Each of these functions is given the whole data set output by f . It is left to the functions themselves to determine which parts of the data to process. This enables varying strategies:

1. MapReduce, where one function is applied to sets based on keys
2. Key-dependent functions are applied to all of the data
3. A mix of the two (possibly even within one processing chain)

The crucial factor for making this parallel is the map in line 8. It is here that multiple instances of the transformer are created, one per key. Therefore in practical implementations, map can be replaced with some parallel processing mechanism.

Relation to MapReduce

I said above that a method $wrapMR$ was needed for wrapping a traditional transformer into the monadic framework. The following code does just that, turning a transformer into a monadic function:

```

wrapMR :: Eq a => ([s] → [(s', b)]) → (a → MapReduce s a s' b)
wrapMR f = λk → MR (g k)
  where
    g k ss = f ∘ map fst ∘ filter (λs → k ≡ snd s) $ ss

```

Note that the transformer need know nothing about the monad. It is simply a function that maps a list of values to a list of value/key pairs. All the business of selecting values for the attention of the transformer is done by $wrapMR$.

Limitations

Composition sends the entire dataset to every thread of execution. This may be desirable in cases where generalised MapReduce is being used, but for standard MapReduce with large datasets on many processors, this may be an issue. Possible approaches involve use of shared memory or disk to store the datasets, or a more carefully engineered implementation of $wrapMR$.

Code

Library

Listings 1–3 show an implementation of generalised MapReduce that is multi-threaded on a multi-processor system with a single OS image. The library consists of three modules: a hashing function for data (Listing 1), a parallel generalisation of *map* (Listing 2), and the *MapReduce* monad itself (Listing 3). Clearly, more complex approaches to parallelisation could be used: for example, coordinating processes running on a cluster of processing nodes.

```

1  {-# LANGUAGE TypeSynonymInstances #-}
2  module Hashable (Hashable, hash) where
3  import qualified Crypto.Hash.MD5 as H
4  import Data.Char (ord)
5  import Data.ByteString (ByteString, pack, unpack)
6  -- any type that can be converted to ByteString is hashable
7  class Hashable s where
8  conv :: s -> ByteString
9  -- the hash function
10 hash :: (Hashable s) => s -> Int
11 hash s = sum $ map fromIntegral (unpack h)
12 where
13   h = H.hash $ conv s
14 -- make String Hashable
15 instance Hashable String where
16 conv s = pack $ map (fromIntegral o ord) s

```

Listing 1: The *Hashable* module

Application

Listing 4 shows an implementation of the classic first MapReduce program: a word-count routine. Note that the MapReduce part of the application (lines 14–17) requires only two lines of active code. Also, the transformers are pure data processing; they are completely unaware of the generalised MapReduce framework. Finally, observe the use of *distributeMR* to spread the initial data across

```
1  module ParallelMap (map) where
2  import Control.Parallel.Strategies (parMap, rdeepseq)
3  import Control.DeepSeq (NFData)
4  import Prelude hiding (map)
5  map :: (NFData b) => (a -> b) -> [a] -> [b]
6  map = parMap rdeepseq
```

Listing 2: The *ParallelMap* module

multiple mappers. We have established an extremely elegant architecture in which transformations can be plugged into a parallel processing chain with no difficulty.

Performance

The application has been tested on a heavily loaded dual core MacBook Pro on datasets of up to 1,000,000 words. Specifically, the application was tested varying the following parameters:

1. Number of words: 10000, 100000, 250000, 500000, 1000000
2. Size of vocabulary: 500, 1000, 2000
3. Number of cores used: 1, 2

As can be seen from Figure 1, on a single core, performance scaled approximately linearly with corpus size at a rate of approximately 100,000 words per second. On two cores performance was noticeably worse, and was distinctly non-linear, also depending on the vocabulary size.

I suspect the poor performance on two cores was due to the data passing issue discussed above, with bus and memory conflicts coming to the fore. Note, however that *this code has no optimisations* having been written for clarity rather than performance. Therefore it is expected that considerably better results could be obtained with very little effort.

```

1  module MapReduce
2      (MapReduce, return, ( $\gg$ ), runMapReduce, distributeMR, wrapMR
3      ) where
4  import Data.List (nub)
5  import Control.DeepSeq (NFData)
6  import Prelude hiding (return, ( $\gg$ ))
7  import Hashable (Hashable, hash)
8  import qualified ParallelMap as P
9  class Monad' m where
10     return :: a → m s x s a
11     ( $\gg$ ) :: (Eq b, NFData s'', NFData c) ⇒
12             m s a s' b → (b → m s' b s'' c) → m s a s'' c
13 newtype MapReduce s a s' b = MR {runMR :: [(s, a)] → [(s', b)]}
14 instance Monad' MapReduce where
15     return = retMR
16     ( $\gg$ ) = bindMR
17 retMR :: a → MapReduce s x s a
18 retMR k = MR $ \ss → [(s, k) | (s, _) ← ss]
19 bindMR :: (Eq b, NFData s'', NFData c) ⇒
20           MapReduce s a s' b →
21           (b → MapReduce s' b s'' c) → MapReduce s a s'' c
22 bindMR f g = MR $ \s →
23     let fs = runMR f s
24         gs = map g ∘ nub ∘ map snd $ fs
25     in concat $ P.map ( $\lambda$ g' → runMR g' fs) gs
26     -- execute a MapReduce action given an initial state
27 runMapReduce :: MapReduce s () s' b → [s] → [(s', b)]
28 runMapReduce m ss = runMR m [(s, ()) | s ← ss]
29     -- distribute a data set across a range of keys
30 distributeMR :: (Hashable s) ⇒ MapReduce s () s Int
31 distributeMR = MR $ \ss → [(s, hash s) | (s, _) ← ss]
32     -- wrap a transformer in a monadic function
33 wrapMR :: (Eq a) ⇒ ([s] → [(s', b)] → (a → MapReduce s a s' b)
34 wrapMR f =  $\lambda$ k → MR (g k)
35     where
36         g k ss = f ∘ map fst ∘ filter ( $\lambda$ s → k ≡ snd s) $ ss

```

Listing 3: The MapReduce Module

```

1  module Main where
2  import MapReduce
3  import System.IO
4  import System.Environment (getArgs)
5  import Prelude hiding (return, ( $\gg$ ))
6  import qualified Prelude as P
7  main :: IO ()
8  main = do
9      args ← getArgs
10     state ← getLines (head args)
11     let res = mapReduce state
12     putStrLn $ show res
13     -- perform MapReduce
14     mapReduce :: [String] → [(String, Int)]
15     mapReduce state = runMapReduce mr state
16     where
17         mr = distributeMR  $\gg$  wrapMR mapper  $\gg$  wrapMR reducer
18         -- transformers
19         mapper :: [String] → [(String, String)]
20         mapper [] = []
21         mapper (x : xs) = parse x  $\#$  mapper xs
22         where
23             parse x = map ( $\lambda w \rightarrow (w, w)$ ) $ words x
24         reducer :: [String] → [(String, Int)]
25         reducer [] = []
26         reducer xs = [(head xs, length xs)]
27         -- get input
28     getLines :: FilePath → IO [String]
29     getLines file = do
30         h ← openFile file ReadMode
31         text ← hGetContents h
32         P.return (lines text)

```

Listing 4: Word count MapReduce application

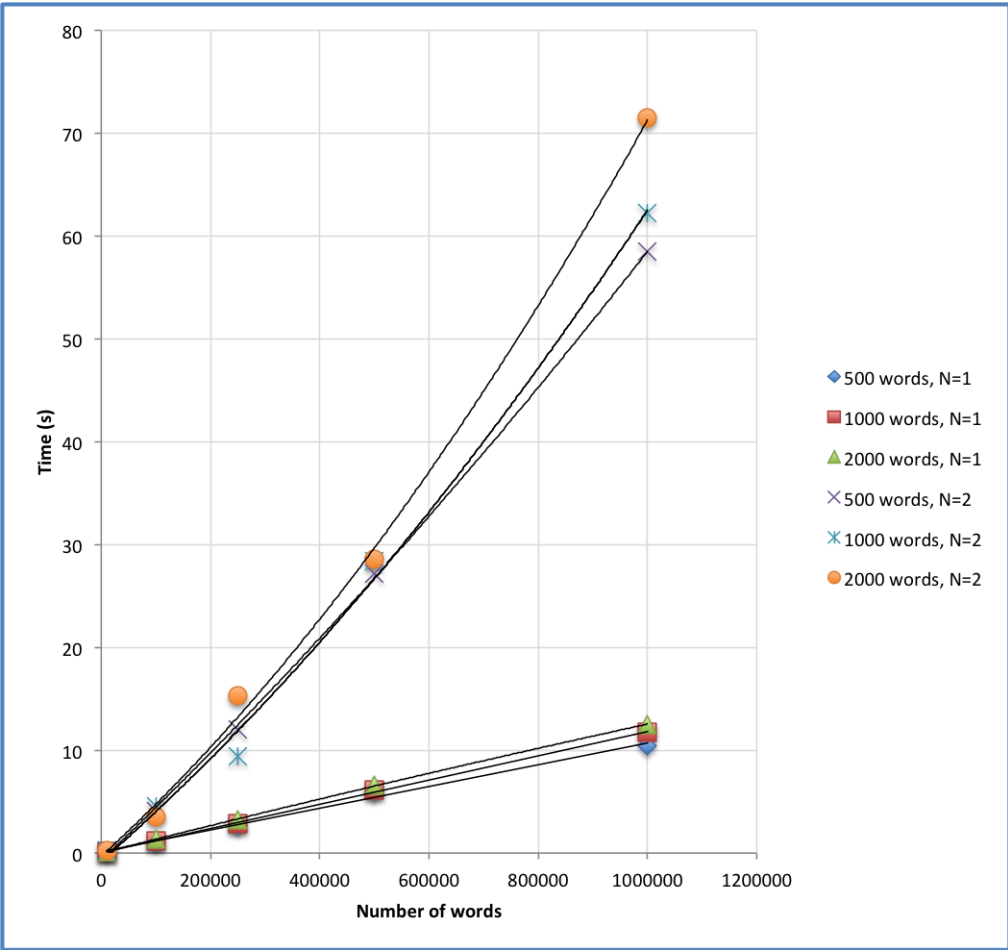


Figure 1: Performance of the application

Fun with Parallel Monad Comprehensions

by Tomas Petricek <tomas.petricek@cl.cam.ac.uk>

Monad comprehensions have an interesting history. They were the first syntactic extension for programming with monads. They were implemented in Haskell, but later replaced with plain list comprehensions and monadic `do` notation. Now, monad comprehensions are back in Haskell, more powerful than ever before!

Redesigned monad comprehensions generalize the syntax for working with lists. Quite interestingly, they also generalize syntax for zipping, grouping and ordering of lists. This article shows how to use some of the new expressive power when working with well-known monads. You'll learn what "parallel composition" means for parsers, a poor man's concurrency monad and an evaluation order monad.

Introduction

This article is inspired by my earlier work on *joinads* [1], an extension that adds pattern matching on abstract values to the *computation expression* syntax in F#. Computation expressions are quite similar to the `do` notation in Haskell. After implementing the F# version of *joinads*, I wanted to see how the ideas would look in Haskell. I was quite surprised to find out that a recent extension for GHC adds some of the expressive power of *joinads* to Haskell.

To add some background: the F# computation expression syntax can be used to work with *monads*, but also with *monoids* and a few other abstract notions of computation. It also adds several constructs that generalize imperative features of F#, including `while` and `for` loops as well as exception handling. The *joinads* extension adds support for pattern-matching on "monadic values". For example, you can define a parallel programming monad and use *joinads* to wait until two parallel computations both complete or wait until the first of the two completes returning a value matching a particular pattern.

How are F# joinads related to Haskell? A recent GHC patch implemented by Nils Schweinsberg [2, 3] brings back support for monad comprehensions to Haskell. The change is now a part of the main branch and will be available in GHC starting with the 7.2 release. The patch doesn't just re-implement original monad comprehensions, but also generalizes recent additions to list comprehensions, allowing parallel monad comprehensions and monadic versions of operations like ordering and grouping [4].

The operation that generalizes parallel comprehensions is closely related to a `merge` operation that I designed for F# joinads. In the rest of this article, I demonstrate some of the interesting programs that can be written using this operation and the elegant syntax provided by the re-designed monad comprehensions.

Quick review of list comprehensions

List comprehensions are a very powerful mechanism for working with lists in Haskell. I expect that you're already familiar with them, but let me start with a few examples. I will use the examples later to demonstrate how the generalized monad comprehension syntax works in a few interesting cases.

If we have a list `animals` containing "cat" and "dog" and a list `sounds` containing animal sounds "meow" and "woof", we can write the following snippets:

```
> [ a ++ " " ++ s | a <- animals, s <- sounds ]
["cat meow", "cat woof", "dog meow", "dog woof"]
```

```
> [ a ++ " " ++ s | a <- animals, s <- sounds, a !! 1 == s !! 1 ]
["dog woof"]
```

```
> [ a ++ " " ++ s | a <- animals | s <- sounds ]
["cat meow", "dog woof"]
```

The first example uses just the basic list comprehension syntax. It uses two *generators* to implement a Cartesian product of the two collections. The second example adds a guard to specify that we want only pairs of strings whose second character is the same. The guard serves as an additional filter for the results.

The last example uses parallel list comprehensions. The syntax is available after enabling the `ParallelListComp` language extension. It allows us to take elements from multiple lists, so that the n^{th} element of the first list is matched with the n^{th} element of the second list. The same functionality can be easily expressed using the `zip` function.

Generalizing to monad comprehensions

The three examples we've seen in the previous section are straightforward when working with lists. After installing the latest development snapshot of GHC and turning on the `MonadComprehensions` language extension, we can use the same syntax for working with further notions of computation. If we instantiate the appropriate type classes, we can even use guards, parallel comprehensions and operations like ordering or grouping. Figure 1 shows some of the type classes and functions that are used by the desugaring.

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a

class (Monad m) => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

class (Monad m) => MonadZip m where
  mzip :: m a -> m b -> m (a, b)

guard :: MonadPlus m => Bool -> m ()
guard b = if b then return () else mzero
```

Figure 1: Type classes and functions used by monad comprehensions

Aside from `Monad`, the desugaring also uses the `MonadPlus` and `MonadZip` type classes. The former is used only for the `guard` function, which is also defined in Figure 1. The latter is a new class which has been added as a generalization of parallel list comprehensions. The name of the function makes it clear that the type class is a generalization of the `zip` function. The patch also defines a `MonadGroup` type class that generalizes grouping operations inside list comprehensions, but I will not discuss that feature in this article.

You can find the general desugaring rules in the patch description [2]. In this article, we'll just go through the examples from the previous section and examine what the translation looks like. The following declaration shows how to implement the `Monad`, `MonadPlus`, and `MonadZip` type classes for lists:

```
instance Monad [] where
  source >>= f = concat $ map f source
  return a = [a]
```

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)

instance MonadZip [] where
  mzip = zip
```

The `>>=` operation, called `bind`, applies the provided function to each element of the input list and then concatenates the generated lists. The `return` function creates a singleton list containing the specified value. The `mzero` value from `MonadPlus` type class is an empty list, which means that `guard` returns `[]` when the argument is `True` and the empty list otherwise. Finally, the `mzip` function for lists is just `zip`.

Now we have everything we need to look at the desugaring of monad comprehensions. The first example from the previous section used multiple generators and can be translated purely in terms of `Monad`:

```
animals >>= (\a -> sounds >>= (\s ->
  return $ a ++ " " ++ b))
```

Every generator is translated to a binding using `>>=`. The operations are nested, and the innermost operation always returns the result of the output function. The next snippet shows what happens when we add a predicate to filter the results:

```
animals >>= (\a -> sounds >>= (\s ->
  guard (a !! 1 == s !! 1) >>= (\_ ->
    return $ a ++ " " ++ s) ))
```

A predicate is translated into a call to the `guard` function in the innermost part of the desugared expression. When the function returns `mzero` value (an empty list), the result of the binding will also be `mzero`, so the element for which the predicate doesn't hold will be filtered out. Finally, let's look at the translation of the last example:

```
(animals 'mzip' sounds) >>= (\(a, s) ->
  return $ a ++ " " ++ s)
```

When we use parallel comprehensions, the inputs of the generators are combined using the `mzip` function. The result is passed to the `bind` operation, which applies the output function to values of the combined computation. If we also specified filtering, the `guard` function would be added to the innermost expression, as in the previous example.

As you can see, the translation of monad comprehensions is quite simple, but it adds expressivity to the syntax for working with monads. In particular, the `do` notation doesn't provide an equivalent syntactic extension for writing parallel comprehensions. (Constructs like generalized ordering, using functions of type `m a -> m a`, and generalized grouping, using functions of type `m a -> m (m a)`, add even more expressivity, but that's a topic for another article.) In the next three sections, I show how we could implement the `mzip` operation for several interesting monads, representing parsers, resumable computations, and parallel computations. At the end of the article, I also briefly consider laws that should hold about the `mzip` operation.

Composing parsers in parallel

What does a *parallel composition of two parsers* mean? Probably the best thing we can do is to run both parsers on the input string and return a tuple with the two results. That sounds quite simple, but what is this construct good for? Let's first implement it and then look at some examples.

Introducing parsers

A parser is a function that takes an input string and returns a list of possible results. It may be empty (if the parser fails) or contain several items (if there are multiple ways to parse the input). The implementation I use in this article mostly follows the one by Hutton and Meijer [5].

```
newtype Parser a
  = Parser (String -> [(a, Int, String)])
```

The result of parsing is a tuple containing a value of type `a` produced by the parser, the number of characters consumed by the parser, and the remaining unparsed part of the string. The `Int` value represents the number of characters consumed by the parser. It is not usually included in the definition, but we'll need it in the implementation of `mzip`.

Now that we have a definition of parsers, we can create our first primitive parser and a function that runs a parser on an input string and returns the results:

```
item :: Parser Char
item = Parser (\input -> case input of
  ""    -> []
  c:cs -> [(c, 1, cs)])
```

```
run :: Parser a -> [a]
run (Parser p) input =
  [ result | (result, _, tail) <- p input, tail == [] ]
```

The `item` parser returns the first character of the input string. When it succeeds, it consumes a single character, so it returns 1 as the second element of the tuple. The `run` function applies the underlying function of the parser to a specified input. As specified by the condition `tail == []`, the function returns the results of those parsers which parsed the entire input. The next step is to make the parser monadic.

Implementing the parser monad

Parsers are well known examples of *monads* and of *monoids*. This means that we can implement both the `Monad` and the `MonadPlus` type classes for our `Parser` type. You can find the implementation in Figure 2.

```
instance Monad Parser where
  return a = Parser (\input -> [(a, 0, input)])
  (Parser p1) >>= f = Parser (\input ->
    [ (result, n1 + n2, tail)
      | (a, n1, input') <- p1 input
        , let (Parser p2) = f a
          , (result, n2, tail) <- p2 input' ])

instance MonadPlus Parser where
  mzero = Parser (\_ -> [])
  mplus (Parser p1) (Parser p2) = Parser (\input ->
    p1 input ++ p2 input)
```

Figure 2: Instances of `Monad` and `MonadPlus` for parsers

The `return` operation returns a single result containing the specified value that doesn't consume any input. The `>>=` operation can be implemented using ordinary list comprehensions. It runs the parsers in sequence, returns the result of the second parser and consumes the sum of characters consumed by the first and the second parser. The `mzero` operation creates a parser that always fails, and `mplus` represents a nondeterministic choice between two parsers.

The two type class instances allow us to use some of the monad comprehension syntax. We can now use the `item` primitive to write a few simple parsers:

```
sat :: (Char -> Bool) -> Parser Char
```

```
sat pred = [ ch | ch <- item, pred ch ]
```

```
char, notChar :: Char -> Parser Char
char ch      = sat (ch ==)
notChar ch   = sat (ch /=)
```

```
some p = [ a:as | a <- p, as <- many p ]
many p = some p 'mplus' return []
```

The `sat` function creates a parser that parses a character matching the specified predicate. The *generator syntax* `ch <- item` corresponds to monadic binding and is desugared into an application of the `>>=` operation. Because the `Parser` type is an instance of `MonadPlus`, we can use the predicate `pred ch` as a guard. The desugared version of the function is:

```
sat pred = item >>= (\ch ->
  guard (pred ch) >>= (\_ -> return ch))
```

The `some` and `many` combinators are mutually recursive. The first creates a parser that parses one or more occurrences of `p`. We encode it using a monad comprehension with two bindings. The parser parses `p` followed by `many p`. Another way to write the `some` parser would be to use combinators for working with applicative functors. This would allow us to write just `(:) <$> p <*> many p`. However, using combinators becomes more difficult when we need to specify a guard as in the `sat` parser. Monad comprehensions provide a uniform and succinct alternative.

The order of monadic bindings usually matters. The monad comprehension syntax makes this fact perhaps slightly less obvious than the `do` notation. To demonstrate this, let's look at a parser that parses the body of an expression enclosed in brackets:

```
brackets :: Char -> Char -> Parser a -> Parser a
brackets op cl body =
  [ inner
    | _ <- char op
    , inner <- brackets op cl body 'mplus' body
    , _ <- char cl ]
```

```
skipBrackets = brackets '(' ')' (many item)
```

The `brackets` combinator takes characters representing opening and closing brackets and a parser for parsing the body inside the brackets. It uses a monad comprehension with three binding expressions that parse an opening brace, the body or more brackets, and then the closing brace.

If you run the parser using `run skipBrackets "((42))"` you get a list containing "42", but also "(42)". This is because the `many item` parser can also consume brackets. To correct that, we need to write a parser that accepts any character except opening and closing brace. As we will see shortly, this can be elegantly solved using parallel comprehensions.

Parallel composition of parsers

To support parallel monad comprehensions, we need to implement `MonadZip`. As a reminder, the type class defines an operation `mzip` with the following type:

```
mzip :: m a -> m b -> m (a, b)
```

By looking just at the type signature, you can see that the operation can be implemented in terms of `>>=` and `return` like this:

```
mzip ma mb = ma >>= \a -> mb >>= \b -> return (a, b)
```

This is a reasonable definition for some monads, such as the `Reader` monad, but not for all of them. For example, `mzip` for lists should be `zip`, but the definition above would behave as a Cartesian product! A more interesting definition for parsers, which cannot be expressed using other monad primitives, is parallel composition from Figure 3.

```
instance MonadZip Parser where
  mzip (Parser p1) (Parser p2) = Parser (\input ->
    [ ((a, b), n1, tail1)
      | (a, n1, tail1) <- p1 input
        , (b, n2, tail2) <- p2 input
        , n1 == n2 ])
```

Figure 3: Instance of `MonadZip` type class for parsers

The parser created by `mzip` independently parses the input string using both of the parsers. It uses list comprehensions to find all combinations of results such that the number of characters consumed by the two parsers was the same. For each matching combination, the parser returns a tuple with the two parsing results. Requiring that the two parsers consume the same number of characters is not an arbitrary decision. It means that the remaining unconsumed strings `tail1` and `tail2` are the same and so we can return either of them. Using a counter is more efficient than comparing strings and it also enables working with infinite strings.

Let's get back to the example with parsing brackets. The following snippet uses parallel monad comprehensions to create a version that consumes all brackets:


```
skipAllBrackets = brackets '(' ')' body
  where body = many [ c | c <- notChar '(' | _ <- notChar ')' ]
```

The parser `body` takes zero or more of any characters that are not opening or closing brackets. The parallel comprehension runs two `notChar` parsers on the same input. They both read a single character and they succeed if the character is not `'(` and `')` respectively. The resulting parser succeeds only if both of them succeed. Both parsers return the same character, so we return the first one as the result and ignore the second.

Another example where this syntax is useful is validation of inputs. For example, a valid Cambridge phone number consists of 10 symbols, contains only digits, and starts with 1223. The new syntax allows us to directly encode these three rules:

```
cambridgePhone =
  [ n | n <- many (sat isDigit)
    | _ <- replicateM 10 item
    | _ <- startsWith (string "1223") ]
```

The encoding is quite straightforward. We need some additional combinators, such as `replicateM`, which repeats a parser a specified number of times, and `startsWith`, which runs a parser and then consumes any number of characters.

We could construct a single parser that recognizes valid Cambridge phone numbers without using `mzip`. The point of this example is that we can quite nicely combine several independent rules, which makes the validation code easy to understand and extend.

Parallel composition of context-free parsers

Monadic parser combinators are very expressive. In fact, they are often *too* expressive, which makes it difficult to implement the combinators efficiently. This was a motivation for the development of non-monadic parsers, such as the one by Swierstra [6, 7], which are less expressive but more efficient. *Applicative functors*, developed by McBride and Paterson [8], are a weaker abstraction that can be used for writing parsers. Figure 4 shows the Haskell type class `Applicative` that represents applicative functors.

If you're familiar with applicative functors, you may know that there is an alternative definition of `Applicative` that uses an operation with the same type signature as `mzip`. We could use `mzip` to define an `Applicative` instance, but this would give us a very different parser definition! In some sense, the following example combines two different applicative functors, but I'll write more about that in the next section.

```
class (Functor f) => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Figure 4: Definition of the `Applicative` type class

The usual applicative parsers allow us to write parsers where the choice of the next parser doesn't depend on the value parsed so far. In terms of formal language theory, they can express only *context-free* languages. This is still sufficient for many practical purposes. For example, our earlier `brackets` parser can be written using the applicative combinators:

```
brackets op cl body =
  pure (\_ inner _ -> inner)
  <*> char op
  <*> brackets op cl body 'mplus' body
  <*> char cl
```

The example first creates a parser that always succeeds and returns a function using the `pure` combinator. Then it applies this function (contained in a parser) to three arguments (produced by the three parsers). The details are not important, but the example shows that comprehensions with independent generators can be expressed just using the `Applicative` interface.

The interesting question is, what operation does `mzip` represent for context-free grammars? A language we obtain if parses for two other languages both succeed is an *intersection* of the two languages. An intersection of two context-free languages is not necessarily context-free, which can be demonstrated using the following example:

$$A = \{a^m b^m c^n \mid m, n \geq 0\} \quad B = \{a^n b^m c^m \mid m, n \geq 0\}$$

$$A \cap B = \{a^m b^m c^m \mid m \geq 0\}$$

The language A accepts words that start with some number of 'a' followed by the same number of 'b' and then arbitrary number of 'c' symbols. The language B is similar, but it starts with a group of any length followed by two groups of the same length. Both are context-free. In fact, our parser `brackets` can be used to parse the two character groups with the same length.

The intersection $A \cap B$ is not context-free [9], but we can easily encode it using parallel composition of parsers. We don't need the full power of monads to parse the first group and calculate its length. It could be implemented just in terms of `Applicative` and `mzip`, but we use the nicer monad comprehension syntax:

```
[ True | _ <- many $ char 'a', _ <- brackets 'b' 'c' unit
      | _ <- brackets 'a' 'b' unit, _ <- many $ char 'c' ]
where unit = return ()
```

The example uses both parallel and sequential binding, but the sequential composition doesn't involve dependencies on previously parsed results. It uses `brackets` to parse two groups of the same length, followed (or preceded) by `many` to consume the remaining group of arbitrary length.

Applicative and parallel parsers

Before moving to the next example, let's take a look how `mzip` relates to applicative functors. As already mentioned, an alternative definition of applicative functors ([8] section 7) uses an operation with exactly the same type signature as `mzip`. You can find the alternative definition in Figure 5.

```
class Functor f => Monoidal f where
  unit  :: f ()
  (★)   :: f a -> f b -> f (a, b)

pure :: Monoidal f => a -> f a
pure a = fmap (const a) unit

(<*>) :: Monoidal f => f (a -> b) -> f a -> f b
f <*> a = fmap (uncurry ($)) (f ★ a)
```

Figure 5: Alternative definition of Applicative operations

Applicative functors are more general than monads, which means that every monad is also an applicative functor. When introducing the `mzip` operation, we attempted to define it in terms of `return` and `>>=`. That code was actually a definition of `★`. However, as already explained, we cannot use that definition (or the `★` operation) for `mzip`, because that definition isn't always intuitively right. Referring to intuition is always tricky, so I'll express my expectations more formally in terms of laws at the end of the article. However, if we always just used `★` as the `mzip` operation, we wouldn't get any additional expressive power, so there would be no point in using parallel monad comprehensions. The expression `[e | a <- e1 | b <- e2]` would mean exactly the same thing as `[e | a <- e1, b <- e2]`.

For example, the definition of `★` for the usual `List` monad gives us the Cartesian product of lists. This isn't very useful, because we can get that behavior using

multiple generators. Instead, parallel list comprehensions use zipping of lists, which comes from a different applicative functor, namely `ZipList`.

The example with parsers is similar. The implementation of `★` would give us sequential composition of parsers. This wouldn't be very useful, so I defined `mzip` as the intersection of parsers. This is an interesting operation that adds expressive power to the language. The `mzip` operation also defines an instance of applicative functor for parsers, but a different one.

For the fans of category theory, the `★` operation is a natural transformation of a *monoidal functor* that can be defined by the monad we're working with. The `mzip` operation can be also viewed as a natural transformation of some monoidal functor, but it may be a different one. One of the additional laws that we can require about `mzip` is commutativity (more about that later), which means that `mzip` should be defined by a *symmetric monoidal functor*.

Parallelizing cooperative computations

As the name *parallel* monad comprehensions suggests, we can use the syntax for running computations in parallel. Unlike comprehensions with multiple generators, parallel comprehensions cannot have any dependencies between the composed bindings. This means that the computations can be evaluated independently.

In this section, I demonstrate the idea using a poor man's concurrency monad inspired by Claessen [10]. The monad can be used to implement a lightweight cooperative concurrency. When running two computations in parallel, we can allow interleaving of atomic actions from the two threads.

Modelling resumable computations

The example I demonstrate here models computations using *resumptions*. This concept is slightly simpler than the original poor man's concurrency monad (which is based on continuations). A resumption is a computation that has either finished and produced some value or has not finished, in which case it can run one atomic step and produce a new resumption:

```
data Monad m => Resumption m r
  = Step (m (Resumption m r))
  | Done r
```

The type is parametrized over a monad and a return type. When evaluating a resumption, we repeatedly run the computation step-by-step. While evaluating, we perform the effects allowed by the monad `m` until we eventually get a result of type `r`. If you're interested in more details about the `Resumption` type, you can find

similar definitions in Harrison’s cheap threads [11] and Papaspyrou’s resumption transformer [12].

Now that we have a type, we can write a function to create a `Resumption` that runs a single atomic action and then completes, and a function that runs a `Resumption`:

```
run :: Monad m => Resumption m r -> m r
run (Done r) = return r
run (Step m) = m >>= run

action :: Monad m => m r -> Resumption m r
action a = Step [ Done r | r <- a ]
```

The `run` function takes a resumption that may perform effects specified by the monad `m` and runs the resumption inside the monad until it reaches `Done`. The function runs the whole computation sequentially (and it cannot be implemented differently). The cooperative concurrency can be added later by creating a combinator that interleaves the steps of two `Resumption` computations.

The `action` function is quite simple. It returns a `Step` that runs the specified action inside the monad `m` and then wraps the result using the `Done` constructor. I implemented the function using monad comprehensions to demonstrate the notation again, but it could be equally written using combinators or the `do` notation.

Implementing the resumption monad

You can see the implementation of `Monad` instance for `Resumption m` in Figure 6. The `return` operation creates a new resumption in the “done” state which contains the specified value. The `>>=` operation constructs a resumption that gets the result of the first resumption and then calls the function `f`. When the left parameter is `Done`, we apply the function to the result and wrap the application inside `return` (because the function is pure) and `Step`. When the left parameter is `Step`, we create a resumption that runs the step and then uses `>>=` recursively to continue running steps from the left argument until it finishes.

The listing also defines an instance of `MonadTrans` to make `Resumption` a monad transformer. The `lift` function takes a computation in the monad `m` and turns it into a computation in the `Resumption` monad. This is exactly what our function for wrapping atomic actions does.

Equipped with the two type class instances and the `run` function, we can write some interesting computations. The following function creates a computation that runs for the specified number of steps, prints some string in each step and then returns a specified value:

```

instance Monad m => Monad (Resumption m) where
  return a = Done a
  (Done r) >>= f = Step $ return (f r)
  (Step s) >>= f = Step $ do
    next <- s
    return $ next >>= f

instance MonadTrans Resumption where
  lift = action

```

Figure 6: Instances of `Monad` and `MonadTrans` for resumptions.

```

printLoop :: String -> Int -> a -> Resumption IO a
printLoop str count result = do
  lift $ putStrLn str
  if count == 1 then return result
  else printLoop str (count - 1) result

cats = run $ printLoop "meow" 3 "cat"

```

The function is written using the `do` notation. It first prints the specified string, using `lift` to turn the `IO ()` action into a single-step `Resumption IO ()`. When the counter reaches one, it returns the result; otherwise it continues looping.

The snippet defines a simple computation `cats` that prints “meow” three times and then returns a string “cat”. The computations created by `printLoop` are not fully opaque. If we have two computations like `cats`, we can treat them as sequences of steps and interleave them. This is what the `mzip` operation does.

Parallel composition of resumptions

If we have two resumptions, we can compose them to run in sequence either using the `do` notation or using a monad comprehension with two generators. To run them in parallel, we need to implement interleaving of the steps as shown in Figure 7.

The result of `mzip` is a resumption that consists of multiple steps. In each step, it performs one step of both of the resumptions given as arguments. When it reaches a state when both of the resumptions complete and produce results, it returns a tuple containing the results using `Done`. A step is performed using an effectful `step` function. To keep the implementation simple, we keep applying `step` to both of the resumptions and then recursively combine the results. Applying `step` to a

```
instance Monad m => MonadZip (Resumption m) where
  mzip (Done a) (Done b) = Done (a, b)
  mzip sa sb = Step [ mzip a b | a <- step sa, b <- step sb ]
  where step (Done r) = return $ Done r
        step (Step sa) = sa
```

Figure 7: Instance of `MonadZip` that composes resumptions in parallel

resumption that has already completed isn't a mistake. This operation doesn't do anything and just returns the original resumption (without performing any effects).

Once we define `mzip`, we can start using the parallel comprehension syntax for working with resumptions. The next snippet demonstrates two ways of composing resumptions. In both examples, we compose a computation that prints “meow” two times and then returns “cat” with a computation that prints “woof” three times and then returns “dog”:

```
animalsSeq =
  [ c ++ " and " ++ d
    | c <- printLoop "meow" 2 "cat"
    , d <- printLoop "woof" 3 "dog" ]
```

```
animalsPar =
  [ c ++ " and " ++ d
    | c <- printLoop "meow" 2 "cat"
    | d <- printLoop "woof" 3 "dog" ]
```

The only difference between the two examples is that the first one composes the operations using multiple generators (separated by comma) and the second one uses parallel comprehensions (separated by bar).

When you run the first example, the program prints meow, meow, woof, woof, woof and then returns a string “cat and dog”. The second program interleaves the steps of the two computations and prints meow, woof, meow, woof, woof and then returns the same string.

Composing computations in parallel

In the previous section, we used the parallel comprehension syntax to create computations that model parallelism using resumptions. Resumptions can be viewed as lightweight cooperative threads. They are useful abstraction, but the simple implementation in the previous section does not give us any speed-up on multi-core

CPU. This section will follow a similar approach, but we look at how to implement actual parallelism based on *evaluation strategies*.

Marlow *et al.* [13] introduced an `Eval` monad for explicitly specifying evaluation order. I will start by briefly introducing the monad, so don't worry if you're not familiar with it already. I'll then demonstrate how to define a `MonadZip` instance for this monad. This way, we can use the parallel comprehension syntax for actually running computations in parallel.

Introducing the evaluation-order monad

The evaluation-order monad is represented by a type `Eval a`. When writing code inside the monad, we can use several functions of type `a -> Eval a` that are called *strategies*. These functions take a value of type `a`, which may be unevaluated, and wrap it inside the monad. A strategy can specify how to evaluate the (unevaluated) value. The two most common strategies are `rpar` and `rseq` (both are functions of type `a -> Eval a`). The `rpar` strategy starts evaluating the value in background and `rseq` evaluates the value eagerly before returning.

A typical pattern is to use the `do` notation to spawn one computation in parallel and then run another computation sequentially. This way we can easily parallelize two function calls:

```
fib38 = runEval $ do
  a <- rpar $ fib 36
  b <- rseq $ fib 37
  return $ a + b
```

The example shows how to calculate the 38th Fibonacci number. It starts calculating `fib 36` in parallel with the rest of the computation and then calculates `fib 37` sequentially. The `do` block creates a value of type `Eval Integer`. We then pass this value to `runEval`, which returns the wrapped value. Because we used the `rpar` and `rseq` combinators when constructing the computation, the returned value will be already evaluated.

However, it is worth noting that the `return` operation of the monad doesn't specify any evaluation order. The function `runEval . return` is just an identity function that doesn't force evaluation of the argument. The evaluation order is specified by additional combinators such as `rpar`.

The `Eval` monad is implemented in the `parallel` package [14] and very well explained in the paper by Marlow *et al.* [13]. You can find the definition of `Eval` and its monad instance in Figure 8. We don't need to know how `rpar` and `rseq` work, so we omit them from the listing.

The `Eval a` type is simple. It just wraps a value of type `a`. The `runEval` function unwraps the value and the `Monad` instance implements composition of

```
data Eval a = Done a

runEval :: Eval a -> a
runEval (Done x) = x

instance Monad Eval where
  return x = Done x
  Done x >>= k = k x
```

Figure 8: Evaluation-order monad `Eval` with a `Monad` instance

computations in the usual way. The power of the monad comes from the evaluation annotations we can add. We transform values of type `a` into values of type `Eval a`; while doing so, we can specify the evaluation strategy. The strategy is usually given using combinators, but if we add an instance of the `MonadZip` class, we can also specify the evaluation order using the monad comprehension syntax.

Specifying parallel evaluation order

The `mzip` operator for the evaluation order monad encapsulates a common pattern that runs two computations in parallel. We've seen an example in the previous section – the first computation is started using `rpar` and the second one is evaluated eagerly in parallel using `rseq`. You can find the implementation in Figure 9.

```
instance MonadZip Eval where
  mzip ea eb = do
    a <- rpar $ runEval ea
    b <- rseq $ runEval eb
    return (a, b)
```

Figure 9: Instance of `MonadZip` for parallelizing tasks

A tricky aspect of `Eval` is that it may represent computations with explicitly specified evaluation order (created, for example, using `rpar`). We can also create computations without specifying evaluation order using `return`. The fact that `return` doesn't evaluate the values makes it possible to implement the `mzip` function, because the required type signature is `Eval a -> Eval b -> Eval (a, b)`.

The two arguments already have to be values of type `Eval`, but we want to specify the evaluation order using `mzip` after creating them. If all `Eval` values were already

evaluated, then the `mzip` operation couldn't have any effect. However, if we create values using `return`, we can then apply `mzip` and specify how they should be evaluated later. This means that `mzip` only works for `Eval` computations created using `return`.

Once we understand this, implementing `mzip` is quite simple. It extracts the underlying (unevaluated) values using `runEval`, specifies the evaluation order using `rpar` and `rseq` and then returns a result of type `Eval (a, b)`, which now carries the evaluation order specification. Let's look how we can write a sequential and parallel version of a snippet that calculates the 38th Fibonacci number:

```
fibTask n = return $ fib n

fib38seq = runEval [ a + b | a <- fibTask 36
                    , b <- fibTask 37 ]
fib38par = runEval [ a + b | a <- fibTask 36
                    | b <- fibTask 37 ]
```

The snippet first declares a helper function `fibTask` that creates a delayed value using the sequential `fib` function and wraps it inside the `Eval` monad without specifying evaluation strategy. Then we can use the function as a source for generators in the monad comprehension syntax. The first example runs the entire computation sequentially – aside from some wrapping and unwrapping, there are no evaluation order specifications. The second example runs the two sub-computations in parallel. The evaluation order annotations are added by the `mzip` function from the desugared parallel comprehension syntax.

To run the program using multiple threads, you need to compile it using GHC with the `-threaded` option. Then you can run the resulting application with command line arguments `+RTS -N2 -RTS`, which specifies that the runtime should use two threads. I measured the performance on a dual-core Intel Core 2 Duo CPU (2.26GHz). The time needed to run the first version was approximately 13 seconds while the second version completes in 9 seconds.

Writing parallel algorithms

The $\sim 1.4\times$ speedup is less than the maximal $2\times$ speedup, because the example parallelizes two calculations that do not take equally long. To generate a better potential for parallelism, we can implement a recursive `pfib` function that splits the computation into two parallel branches recursively until it reaches some threshold:

```
pfib :: Integer -> Eval Integer
pfib n | n <= 35 = return $ fib n
```

```
pfib n = [ a + b | a <- pfib $ n - 1
            | b <- pfib $ n - 2 ]
```

I hope you'll agree that the declaration looks quite neat. A nice consequence of using parallel comprehensions is that we can see which parts of the computation will run in parallel without any syntactic noise. We just replace a comma with a bar to get a parallel version! The compiler also prevents us from trying to parallelize code that cannot run in parallel, because of data dependencies. For example, let's look at the Ackermann function:

```
ack :: Integer -> Integer -> Eval Integer
ack 0 n = return $ n + 1
ack m 0 = ack (m - 1) 1
ack m n = [ a | na <- ack m (n - 1)
            , a <- ack (m - 1) na ]
```

The Ackermann function is a well-known function from computability theory. It is interesting because it grows very fast (as a result, it cannot be expressed using primitive recursion). For example, the value of `ack 4 2` is $2^{65536} - 3$.

We're probably not going to be able to finish the calculation, no matter how many cores our CPU has. However, we can still try to parallelize the function by replacing the two sequential generators with a parallel comprehension:

```
ack m n = [ a | na <- ack m (n - 1)
            | a <- ack (m - 1) na ]
```

If you try compiling this snippet, you get an error message saying `Not in scope: na`. We can easily see what went wrong if we look at the desugared version:

```
((ack m (n - 1)) 'mzip'
 (ack (m - 1) na)) >>= (\(na, a) -> a)
```

The problem is that the second argument to `mzip` attempts to access the value `na` which is defined later. The value is the result of the first expression, so we can access it only after both of the two parallelized operations complete.

In other words, there is a data dependency between the computations that we're trying to parallelize. If we were not using parallel monad comprehensions, we could mistakenly think that we can parallelize the function and write the following:

```
ack m n = runEval $ do
  na <- rpar $ ack m (n - 1)
  a <- rseq $ ack (m - 1) na
  return a
```

This would compile, but it wouldn't run in parallel! The value `na` needs to be evaluated before the second call, so the second call to `ack` will block until the first one completes. This demonstrates a nice aspect of writing parallel computations using the comprehension syntax. Not only that the syntax is elegant, but the desugaring also performs a simple sanity check on our code.

Parallel comprehension laws

I have intentionally postponed the discussion about laws to the end of the article. So far, we have looked at three different implementations of `mzip`. I discussed some of the expectations informally to aid intuition. The type of `mzip` partially specifies how the operation should behave, but not all well-typed implementations are intuitively right.

In my understanding, the laws about `mzip` are still subject to discussion, although some were already proposed in the discussion about the GHC patch [2]. I hope to contribute to the discussion in this section. We first look at the laws that can be motivated by the category theory behind the operation, and then discuss additional laws inspired by the work on $F^\#$ joinads.

Basic laws of parallel bindings

As already briefly mentioned, the `mzip` operation can be viewed as a *natural transformation* defined by some *monoidal functor* [15]. In practice, this means that the `mzip` operation should obey two laws. The first one is usually called *naturality* and it specifies the behavior of `mzip` with respect to the `map` function of a monad (the function can be implemented in terms of `bind` and `return` and corresponds to `liftM` from the Haskell base library). The second law is *associativity*, and we can express it using a helper function `assoc ((a, b), c) = (a, (b, c))`:

$$\text{map } (f \times g) (\text{mzip } a \ b) \equiv \text{mzip } (\text{map } f \ a) \ (\text{map } g \ b) \quad (1)$$

$$\text{mzip } a \ (\text{mzip } b \ c) \equiv \text{map } \text{assoc } (\text{mzip } (\text{mzip } a \ b) \ c) \quad (2)$$

The naturality law (1) specifies that we can change the order of applying `mzip` and `map`. The equation has already been identified as a law in the discussion about the patch [2]. The law is also required by *applicative functors* [8] – this is not surprising as applicative functors are also monoidal.

The *associativity* law (2) is also very desirable. When we write a comprehension such as `[e | a <- m1 | b <- m2 | c <- m3]`, the desugaring first needs to zip two of the three inputs and then zip the third with the result, because `mzip` is a

binary operation. The order of zipping feels like an implementation detail, but if we don't require associativity, it may affect the meaning of our code.

Parallel binding as monoidal functor

A monoidal functor in category theory defines a natural transformation (corresponding to our `mzip`), but also a special value called *unit*. A Haskell representation of monoidal functor is the `Monoidal` type class from Figure 5. For a monoidal functor `f`, the type of unit is `f ()`. Every applicative functor in Haskell has a unit. For example, the unit value for `ZipList` is an infinite list of `()` values. We do not necessarily need to add unit to the definition of `MonadZip`, because it is not needed by the desugaring. However, it is interesting to explore how a special `munit` value would behave.

The laws for monoidal functors specify that if we combine `munit` with any other value using `mzip`, we can recover the original value using `map snd`. In the language of monad comprehensions, the law says that the expression `[e | a <- m]` should be equivalent to `[e | a <- m | () <- munit]`.

It is important to realize that the computation created using monadic `return` isn't the same thing as unit of the monoidal functor associated with `mzip`. For lists, `return ()` creates a singleton list. The `return` operation is unit of a monoidal functor defined by the monad, but the unit associated with `mzip` belongs to a different monoidal functor!

Symmetry of parallel binding

Another sensible requirement for `mzip` (which exists in $F^\#$ joinads) is that reordering of the arguments only changes the order of elements in the resulting tuple. In theory, this means that the *monoidal functor* defining `mzip` is *symmetric*. We can specify the law using a helper function `swap (a, b) = (b, a)`:

$$mzip\ a\ b \equiv map\ swap\ (mzip\ a\ b) \tag{3}$$

The *symmetry* law (3) specifies that `[(x, y) | x <- a | y <- b]` should mean the same thing as `[(x, y) | y <- b | x <- a]`. This may look like a very strong requirement, but it fits quite well with the usual intuition about the `zip` operation and parallel monad comprehensions. The symmetry law holds for lists, parsers and the evaluation order monad. For the poor man's concurrency monad, it holds if we treat effects that occur within a single step of the evaluation as unordered (which may be a reasonable interpretation). For some monads, such as the `State` monad, it is not possible to define a symmetric `mzip` operation.

The three laws that we've seen so far are motivated by the category theory laws for the (*symmetric*) *monoidal functors* that define our `mzip` operation. However,

we also need to relate the functors in some way to the monads with which we are combining them.

Relations with additional operations

The discussion about the patch [2] suggests one more law that relates the `mzip` operation with the `map` operation of the monad, called *information preservation*:

$$\text{map fst (mzip a b)} \equiv a \equiv \text{map snd (mzip b a)} \quad (4)$$

The law specifies that combining a computation with some other computation using `mzip` and then recovering the original form of the value using `map` doesn't lose information. However, this law is a bit tricky. For example, it doesn't hold for lists if a and b are lists of different length, since the `zip` function restricts the length of the result to the length of the shorter list. Similarly, it doesn't hold for parsers (from the first section) that consume a different number of characters.

However, the law expresses an important requirement: when we combine certain computations, it should be possible to recover the original components. A law that is similar to (4) holds for applicative functors, but with a slight twist:

$$\text{map fst (mzip a munit)} \equiv a \equiv \text{map snd (mzip munit a)} \quad (5)$$

Instead of zipping two arbitrary monadic values, the law for applicative functors zips an arbitrary value with `unit`. In case of lists, `unit` is an infinite list, so the law holds. Intuitively, this holds because `unit` has a maximal structure (in this case, the structure is the length of the list).

Ideally, we'd like to say that combining two values with the same structure creates a new value which also has the same structure. There is no way to refer to the "structure" of a monadic value directly, but we can create values with a given structure using `map`. This weaker law holds for both lists and parsers. Additionally, we can describe the case when one of the two computations is `mzero` and when both of them are created using `return`:

$$\text{map fst (mzip a (map f a))} \equiv a \equiv \text{map snd (mzip (map g a) a)} \quad (6)$$

$$\text{map fst (mzip a mzero)} \equiv \text{mzero} \equiv \text{map snd (mzip mzero a)} \quad (7)$$

$$\text{mzip (return a) (return b)} \equiv \text{return (a, b)} \quad (8)$$

The first law (6) is quite similar to the original law (4). The only difference is that instead of zipping with an arbitrary monadic value b , we're zipping a with a value constructed using `map` (for any total functions f and g). This means that the

actual value(s) that the monadic value contains (or produces) can be different, but the structure will be the same, because `map` is required to preserve the structure.

The second law (7) specifies that `mzero` is the *zero element* with respect to `mzip`. It almost seems that it holds of necessity, because `mzero` with type `m a` doesn't contain any value of type `a`, so there is no way we could construct a value of type `(a,b)`. This second law complements the first (6), but it is not difficult to see that it contradicts the original *information preservation* law (4).

Finally, the third law (8) describes how `mzip` works with respect to the monadic `return` operation. This is interesting, because we're relating `mzip` of one applicative functor with the unit of another applicative functor (defined by the monad). The law isn't completely arbitrary; an equation with a similar structure is required for *causal commutative arrows* [16]. In terms of monad comprehension syntax, the law says that `[e | a <- return e1 | b <- return e2]` is equivalent to `[e | (a, b) <- return (e1, e2)]`.

I believe that the three laws I proposed in this section partly answer the question of how to relate the two structures combined by parallel monad comprehensions – the `MonadPlus` type class with the symmetric monoidal functor that defines `mzip`.

Conclusions

After some time, monad comprehensions are back in Haskell! The recent GHC patch makes them even more useful by generalizing additional features of list comprehensions including parallel binding and support for operations like ordering and grouping. In this article, I focused on the first generalization, although the remaining two are equally interesting.

We looked at three examples: parallel composition of parsers (which applies parsers to the same input), parallel composition of resumptions (which interleaves the steps of computations) and parallel composition of an evaluation order monad (which runs two computations in parallel). Some of the examples are inspired by my previous work on joinads that add a similar language extension to $F^\#$. The $F^\#$ version includes an operation very similar to `mzip` from the newly included `MonadZip` type class. I also proposed several laws – some of them inspired by category theory and some by my work on $F^\#$ joinads – hoping that this article may contribute to the discussion about the laws required by `MonadZip`.

Acknowledgements

Thanks to Alan Mycroft for inspiring discussion about some of the monads demonstrated in this article and to Dominic Orchard for many useful comments on a draft of the article as well as discussion about category theory and the `mzip` laws. I'm

also grateful to Brent Yorgey for proofreading the article and suggesting numerous improvements.

References

- [1] Tomas Petricek and Don Syme. Joinads: A retargetable control-flow construct for reactive, parallel and concurrent programming. In **PADL'11**, pages 205–219 (2011).
- [2] Bring back monad comprehensions. <http://tinyurl.com/ghc4370>.
- [3] Nils Schweinsberg. Fun with monad comprehensions (2010). <http://blog.n-sch.de/2010/11/27/fun-with-monad-comprehensions/>.
- [4] Simon L. Peyton Jones and Philip Wadler. Comprehensive comprehensions. In **Haskell'07**, pages 61–72 (2007).
- [5] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. **Journal of Functional Programming**, 8(4):pages 437–444 (July 1998).
- [6] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In **Advanced Functional Programming, Second International School-Tutorial Text**, pages 184–207 (1996).
- [7] S. Doaitse Swierstra. Combinator parsing: A short tutorial. Technical report, Utrecht University (2008). <http://tinyurl.com/parsing-tutorial>.
- [8] Conor McBride and Ross Paterson. Applicative programming with effects. **Journal of Functional Programming**, 18:pages 1–13 (2007).
- [9] Wikipedia. Pumping lemma for context-free languages (2011). http://en.wikipedia.org/wiki/Pumping_lemma_for_context-free_languages.
- [10] Koen Claessen. A poor man's concurrency monad. **Journal of Functional Programming**, 9:pages 313–323 (May 1999).
- [11] William L. Harrison. Cheap (but functional) threads (2008). <http://www.cs.missouri.edu/~harrison/drafts/CheapThreads.pdf>. Submitted for publication.
- [12] Nikolaos S. Papaspyrou. A resumption monad transformer and its applications in the semantics of concurrency. In **3rd Panhellenic Logic Symposium** (2001).
- [13] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. Seq no more: better strategies for parallel haskell. In **Haskell'10**, pages 91–102. ACM (2010).
- [14] <http://hackage.haskell.org/package/parallel>.

- [15] Wikipedia. Monoidal functor (2011). http://en.wikipedia.org/wiki/Monoidal_functor.
- [16] Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In **ICFP'09**, pages 35–46 (2009).

Attributed Variables: Their Uses and One Implementation

by Douglas M. Auclair <dauclair@hotmail.com>

Here's a computational problem that comes up often enough: find a value within some range. You don't know what the value is at the moment, but you will know it, eventually, as the range becomes further restricted by refinements of the constraints or data.

In general, this is an easy problem for Haskell to solve, and Haskell provides more than one way to approach the solution. For example, if we have some x in some domain D , one way is to look at an arbitrary x until we find the one, true x we are looking for, as in the following code snippet:

$$[x \mid x \leftarrow D, \dots \textit{further constraints}]$$

More variables? Interrelated ones? No problem:

$$[(x, y, z) \mid x \leftarrow D, y \leftarrow D, z \leftarrow D, x < y, \dots \textit{further constraints}]$$

Okay, let's take it to the next level. What if the interrelation is such that one variable consumes from the domain, so that value is now unavailable to other variables? Pithily: the variables' values are mutually exclusive.

Well, one way to go about that is to embed a guard for each variable into the list comprehension:

$$[(x, y, z) \mid x \leftarrow D, y \leftarrow D, y \neq x, z \leftarrow D, z \notin [x, y], \dots]$$

The drawback is that this kind of guarded coding is tedious and prone to errors. On top of that, the fact that each guard follows its select causes unnecessary backtracking through the search space¹, especially given that we know the guarded values a priori.

¹Y'all don't mind if I speak logically, and in the logic-programming paradigm, do you?

Well, then, we can automate guarded selection with a state (transformer) monad²:

```

...
f = do x ← choose
      y ← choose
      z ← choose
...
runStateT f [0..9]

```

Okay, excellent! All of this works, and works well... for a fixed, predetermined variable set.

The Problem

Things start to get tricky, however, when one set of constrained values spill over onto another set of constrained values. For example, the results of one generation in a set of cellular automata or genetic (algorithm) pool affect the next set:

```

evalStateT (gen row1guards) domain ≫= λans1 →
  evalStateT (gen (row2guards ans1)) domain ≫= λans2 →
    evalStateT (gen (row3guards ans1 ans2)) domain ≫= λans3 →
      ... etc

```

The state transformer isn't enough by itself, so then what monad is sufficient to model the above functional behavior? This quickly becomes more of an exercise of managing monads around the problem rather than solving the problem itself.³

The above addresses a simple problem, where we know how many variables we are working with. But it gets worse. What if we do not know, a priori, how many variables we need to solve for? The problem space I'm looking at is something like:

For an unknown number of variables, assign each to a domain and appropriately constrain them.

How do I write the above as a list comprehension expression, or as a monadic one? The problem with Haskell variables is that they aren't things that can be

²See my article in *The Monad.Reader*, Issue 11, "MonadPlus: What a Super Monad!" for an implementation of *choose*, etc.

³No, I'm not speaking disparagingly of the monadic programming style. I rather like that style, and use it in my domain language of predicate logic that I layer on top of 'regular' Haskell programming. The beauty of monads, for me, is that they are invisible (in coding them) and powerful (in their effect). When the code becomes more about monads and less about what they are there to do is when I see an opportunity to examine things in a different light.

carried around and reasoned about in and of themselves, and so something so easily accomplished in a Prolog program:

```
% In a list of variables:

kakuro([H|T], Acc, Functor) :-
% assign a value to the variable at the head:
    assign(H),
% and do the same for the rest of the list:
    kakuro(T, [H|Acc], Functor).

% when the list of variables is empty:
kakuro([], Vars, Functor) :-
% construct a goal from the function name and the assigned variables:
    Fn =.. [Functor, Vars],
% and call the goal:
    call(Fn).
```

called with:

```
?- kakuro([A, B, C, D], [], sum_equals_10).
```

or:

```
?- kakuro([X, Y, Z], [], sum_equals_10).
```

... or with a list of any other length, isn't so easily expressible in Haskell. After all, what is the type of a list of unbound variables? Haskell doesn't (easily) allow this, and I was stuck: I didn't know how to proceed using monadic programming or list comprehensions.

A solution

Fortunately for me, I've been studying constraint-based programming, and that research has led me to some papers on attributed variables⁴ which facilitate this programming style.

Attributed variables are variables ranging over a domain that can have attributes attached to them that constrain the range of values. It's a very simple step to go

⁴[http://www.swi-prolog.org/pldoc/doc_for?object=section\(2,'6.1',swi\('/doc/Manual/attvar.html'\)\)](http://www.swi-prolog.org/pldoc/doc_for?object=section(2,'6.1',swi('/doc/Manual/attvar.html')))

from a set of unbound attributed variables to a simple constraint solver by adding attributes to the variables.

More formally, I declare (denotationally) an attributed variable as:

```
data Attribute a = Attribute [Constraint a] (Domain a)
```

where *Constraint* and *Domain* are declared as:

```
newtype Constraint a = Constraint { resolve :: a → Bool }
type Domain a = [a]
```

Or, to put it in words, an attributed variable is an object that has a set of constraining functions over an assigned domain.⁵

The above denotational declaration of attributed variables works, but I find it useful, operationally, to ground an attributed variable once it's reduced to a single value:

```
data Attribute a = Attribute [Constraint a] (Domain a)
                  | Ground a
```

So, that's my declaration.⁷ How is this implemented in practice?

Implementation

An attributed variable is created in a free state over its domain:

⁵My representation of *Domain* is straightforward and workable for most cases, but it hides more than a few compromises (or "sell-outs") beneath the surface. For example, it makes the (gross) assumption that the range of the domain is enumerable. If *Domain* is a subinterval of \mathbb{R} that no longer holds (at all), so then *Domain* shouldn't have a list representation but a pair of bounds. But even bounds are not comprehensive: take, for example, a *Domain* whose members are incomparable.

But these concerns are, in the main, not ones to lose one's head over. For the most part, simple linear programming solves a vast majority of the World's problems,⁶ and those interesting cases are exactly that: interesting cases.

⁶The preface of *How to Solve It: Modern Heuristics* by Zbigniew Michalewicz and David B. Fogel makes this assertion, from hard-won experience in the field. Seconded.

⁷This is one approach to attributed variables, where constraints are applied to the variable over the domain until it is reduced to a single value, at which point the constraints and even the domain become superfluous and simply go away. An advantage to this approach is that a test for groundedness becomes trivial. A disadvantage is that once grounded, the information about that path that got one there is only implicit in the data flow through the algorithm (the stack trace, as it were). Another disadvantage is that an attributed variable, once grounded, automatically rejects constraints. This may lead to unexpected program behavior, because a variable grounded to 5 with a new constraint of $x > 23$ should fail, but my implementation forwards the 5 value in a success state.

```
makeAttrib :: Domain a → Attribute a
makeAttrib list = Attribute [] list
```

As constraints are added, the system “tries to solve” the attributed variable over the set of the constraints:

```
constrain (Attribute constraints dataset) constraint
  = solve (Attribute (Constraint constraint : constraints) dataset)
constrain (Ground x) _ = Ground x
```

And if the solution results in a singleton list, then the attributed variable becomes *Ground*:

```
solve attr@(Attribute _ []) = attr
solve attr@(Attribute constraints list)
  = let result = solve' constraints list
    in case result of
      [x] → Ground x
      _   → Attribute constraints result
```

Of course, the “solution” to a *Ground* attributed variable is itself:

```
solve (Ground x) = Ground x
```

The above code is simply a dispatch along the binary type. The worker-bee, *solve'*, is simply a call to *filter*:

```
solve' constraints = filter (solveConstraints constraints)
solveConstraints :: [Constraint a] → a → Bool
solveConstraints [] _ = True
solveConstraints (f : rest) x = resolve f x ∧ solveConstraints rest x
```

Is there something from the standard libraries to reduce that to a one-liner? I leave that as an exercise to the reader.

And there you have it, a working implementation of attributed variables. I have test functions for groundedness, solvability and the current state of the domain of an active attributed variable, but those are simple helper functions, easily implemented, and provided in the attached source code.

Uses

So, now, what do we have?

We have variables for which we may specify a domain (“loosely” typing a variable) and which we may then constrain. How can we use such a system?

For one, we can write constraint logic programs in Haskell. The above Prolog example has a nearly direct transliteration into Haskell (with the additional benefit of strong typing):

```
kakuro :: [Int] -> Int -> [[Int]]
kakuro list total = nub $ summer (sort $ map trib list) [] total
```

```
trib :: Int -> Attribute Int
trib x | x == 0    = makeAttrib [1..9]
      | otherwise = Ground x
```

```
summer :: [Attribute Int] -> [Int] -> Int -> [[Int]]
summer [] list total = do
  guard $ sum list == total
  return $ sort list
summer (h : t) list total = do
  a <- domain $ constrain h (notin list)
  summer t (a : list) total
```

And with the above code, we can now do the following:

```
*Kakuro> kakuro [0,0,0] 10
[[1,2,7],[1,3,6],[1,4,5],[2,3,5]]
*Kakuro> kakuro [0,0,0,0] 10
[[1,2,3,4]]
```

Note that I use *sort* above, but attributed variables are not (denotationally) ordered types. Operationally, that is: in logical programming, it makes sense to eliminate early (pruning the search space), so if we have *Ground* values, we wish to consume those first. So, to that end, we can declare the following instances:

```
instance (Eq a) => Eq (Attribute a) where
  Ground x == Ground y = x == y
  _        == _        = False
instance (Ord a) => Ord (Attribute a) where
  Ground x <= Ground y = x <= y
  Ground _ <= _       = True
  _        <= _       = False
```

And with those instance declarations, *sort* automatically works so that ground values are handled first. Neat!

Other Uses

So attributed variables can be used to help one solve kakuro puzzles, or to write a sudoku solver.

And I've been inching my way toward attributed variables so they can be applied to symbolic logic puzzles as well, replacing my Whatever type with attributed variables over the enumerated domain.

Solving puzzles is fun, but puzzles are not very interesting or pragmatic examples in and of themselves.

So how about this example?

```
*Stocks> makeAttribute [Buy, Sell, Hold]
```

Here, constraints are input feeds filtered through weighted moving averages and other indicators. Now this is a system of keen interest for me and is “currently under active and intense research”.

Will such a system work or not? I'm not at liberty to say,⁸ but it is an area that appears to fit well in the domain of constraint programming.

Of the problem domains commonly attacked by software projects – workflow analysis, data mining, planning and scheduling, budgeting, forecasting, (war)gaming, modelling and simulation – many seem good candidates to use constraint-based reasoning. Attributed variables give Haskell programmers another way to express programs in these domains.

Summary and Critique

I have presented an approach to problem solving with attributed variables along with a sample implementation. The usefulness of this approach is that it gathers the domain and the constraints of the problem into one type, bringing the pieces of how to go about solving the problem into one place. This may make reasoning about the problem, and how to go about solving it, conceptually easier. Alternatively, attributed variables may be seen as a rather heavy-weight approach to problem solving. The marshaling of the constraints over a domain may seem like excessive work where alternatives exist to solving the problem. For example, Daniel Wagner, on reading the discussion about attributed variables on my blog, provides an implementation of kakuro using lists⁹:

⁸An audit trail of at least a year of data is necessary before any claims can be made.

⁹<http://logicaltypes.blogspot.com/2011/04/attributed-variables-their-uses-and-one.html?showComment=1303411168800#c6201818510121277453>

```

type Attribute a = [a]
type Constraint a = a → Bool
type Domain a = [a]
makeAttribute = id
constrain xs f = filter f xs

```

Note that the helper functions *solve*, *solve'*, and *solveConstraints* are totally unnecessary in this setting. There's no need to recheck that all the values left in your domain still satisfy all the constraints they used to satisfy, thanks to referential transparency.

If you make the other choice regarding ground variables (*i.e.* that they can be revoked when a constraint rules them out), the implementation of *constrain* is even lighter: *constrain* = *flip filter*. The implementation is so lightweight, in fact, that I don't even think I'd write it down—I'd just eliminate calls to *makeAttribute* and write *filter* instead of *constrain*!

```

prioritize :: [Attribute Int] → [Attribute Int]
prioritize = sortBy (comparing length)
kakuro :: [Int] → Int → [[Int]]
kakuro list total = nub $ summer (prioritize $ map trib list) [] total
trib :: Int → Attribute Int
trib 0 = [1..9]
trib x = [x]
summer :: [Attribute Int] → [Int] → Int → [[Int]]
summer [] list total = do
  guard $ sum list ≡ total
  return $ sort list
summer (h : t) list total = do
  a ← constrain h (flip notElem list)
  summer t (a : list) total

```

And, yes, for this example, Daniel's example shines. Attributed variables are a more heavy-weight approach than simply using a list comprehension here. For other areas? That depends on the problem, of course. The disadvantage of list comprehensions is that all the parts of the problem must be brought to one place to build the expression. With attributed variables, the constraints may be added to the variable as they occur, allowing the solution to be approached gradually, across functions, or even across modules.

The list comprehension problem-solving approach is a very powerful one, one which I am happy to use when that approach makes sense. Attributed variables also have something to add as a technique for solving problems in that they call out the domain explicitly and relate the domain to the set of constraints. Furthermore, as the constraints are also explicitly called out, they, too, can be reasoned about or transformed. This ability to manipulate the constraints gives problem-solving with attributed variables a potential for higher-order problem solving.

Future Work

This article explores attributed variables at their lowest level: assigning a domain and then adding constraints to the attributed variables until a result is resolved. At this level, there are several sets of problems that can be solved, but there is more: attributed variables in the same domain may be related to each other, so that, for example, an attributed variable can be declared to be less than the ground value of a second attributed variable, once it is solved. Or, the sum of a set of attributed variables may be equal to another attributed variable.

The implementation presented in this article does not show how attributed variables in the same domain may be related to each other. This is an area of further research that I am actively pursuing. Once relations between attributed variables in the same domain are implemented, then a new level of expressivity becomes available which will allow for a wider range of problems that can be solved using the attributed variable approach.